# Algorithms using Java for Spreadsheet Dependent Cell Recomputation*

Joe Francoeur

jfrancoe@mitre.org

December 3, 2002

**Abstract**

Java implementations of algorithms used by spreadsheets to automatically recompute the set of cells dependent on a changed cell are described using a mathematical model for spreadsheets based on graph theory. These solutions comprise part of a Java API that allows a client application to read, modify, and maintain spreadsheet data without using the spreadsheet application program that produced it. Features of the Java language that successfully improve the running time performance of the algorithms are also described.

# 1   Introduction

This paper describes algorithms for the recomputation of spreadsheet cells. The assumed context for such a recomputation occurs when a cell's value is changed. In general, a cell is dependent on several others for its value as defined by its formula. Thus, to maintain the integrity of the spreadsheet, the reading of a cell value requires the recomputation of this cell once any of the cells on which it depends has changed.

The algorithms of this paper form the basis of *ExcelComp* [9], a Java [24] application program interface (API) written by the author that allows the client application to read a specially formatted Microsoft Excel [4]

---

| Report Documentation Page | | *Form Approved*<br>*OMB No. 0704-0188* |
|---|---|---|
| Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. | | |

| 1. REPORT DATE<br>**03 DEC 2002** | 2. REPORT TYPE | 3. DATES COVERED<br>**00-12-2002 to 00-12-2002** |
|---|---|---|
| 4. TITLE AND SUBTITLE<br>**Algorithms using Java for Spreadsheet Dependent Cell Recomputation** | | 5a. CONTRACT NUMBER |
| | | 5b. GRANT NUMBER |
| | | 5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) | | 5d. PROJECT NUMBER |
| | | 5e. TASK NUMBER |
| | | 5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)<br>**MITRE Corporation,202 Burlington Road,Bedford,MA,01730-1420** | | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | | 10. SPONSOR/MONITOR'S ACRONYM(S) |
| | | 11. SPONSOR/MONITOR'S REPORT NUMBER(S) |
| 12. DISTRIBUTION/AVAILABILITY STATEMENT<br>**Approved for public release; distribution unlimited** | | |
| 13. SUPPLEMENTARY NOTES | | |
| 14. ABSTRACT | | |
| 15. SUBJECT TERMS | | |

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT<br>**unclassified** | b. ABSTRACT<br>**unclassified** | c. THIS PAGE<br>**unclassified** | | **23** | |

(henceforth referred to as "Excel") spreadsheet output file, and then make changes to cell values within the ExcelComp representation of this spreadsheet. Changes to cell values are followed by the automatic recomputation of dependent cell values using ExcelComp methods. ExcelComp thus allows the client programmer to provide its users with both the data and behavior of an existing spreadsheet without the use of the original spreadsheet application program that produced it.

During the development of ExcelComp, it was realized that choices of algorithms to perform cell recomputation involve two principal trade-offs: 1) ease of use, and 2) running time performance. On the one hand, one mode of ExcelComp can simply load a file at run time that represents the spreadsheet, and then provide its services. While this mode is satisfactory for many tasks, it is unsuitable for those that require a large number of cell recomputations to support dynamic updates to real-time outputs, for example, the updating of an on-screen map that depends on thousands of cell recomputations. To support this latter task, a second mode was developed that allows faster cell recomputation at the expense of a less convenient installation procedure for the spreadsheet representation.

These considerations make ExcelComp an efficient, platform- and vendor-independent Java API that provides built-in spreadsheet emulation for application end-users. In particular, end-users are relieved of the burden of conducting their spreadsheet tasks outside the domain of their running application. In addition to the efficiency won by executing spreadsheet tasks natively, ExcelComp also obviates the need for costly additional licenses required for multiple users of the application software that produced the spreadsheet. Being written in the modern Java programming language allows the client programmer to easily integrate ExcelComp's functionality into current software development efforts.

While other descriptions of spreadsheet algorithms are available [16][17], this paper is distinctive in its use of graph theory to improve the reader's ability to visualize the algorithms, and to provide a basis for a proof of algorithm correctness. It also presents solutions that leverage features in the object-oriented Java API that lead to succinct, yet powerful code.

This paper focuses on the subject algorithms and the specific features of the Java language used by ExcelComp that are well-suited for their implementation. Readers interested in a more detailed specification of ExcelComp from the client programmer's perspective may contact the author.

# 2  A Scenario

Before getting into the technical details that comprise this report, it would be helpful to consider a motivational scenario.

Consider a spreadsheet of financial data, where subtotals, interest earned, and a grand total might be some examples of computed quantities that each depend on entries in several cells. Analysts may use such a spreadsheet to play "what-if" games by varying values in cells that will affect some target cell, such as interest earned. The spreadsheet program would then automatically recompute all cells that are dependent on the ones changed. This capability of a spreadsheet is its hallmark, and distinguishes it from a simple table of values that have no computational relationship to one another.

Suppose that a computer program needs this spreadsheet of information and auto-update capability to carry out its tasks. This program is to provide its users with the what-if capability, and therefore requires not only cell values, but also cell formulas. Since it needs to emulate the recomputation function of the spreadsheet, it must implement algorithms that return the same recomputed values as the spreadsheet program. It is these algorithms of dependent cell recomputation that are the subject of this report.

# 3  Modes

ExcelComp has two modes of operation:

**Interpreted mode** requires the reading of an eXtensible Markup Language (XML) [2] representation of a spreadsheet. Once this file is parsed and loaded into ExcelComp's data structures, the subject algorithms are implemented via ExcelComp methods. It is called *interpreted*, because cell formulas are interpreted at run time using a custom parser that recognizes a subset of Excel's formula language.

**Compiled mode** uses cell-specific Java classes, created as an offline preprocessing task, to evaluate a cell by recursively evaluating each child cell referenced in its formula's parse tree.

The interpreted mode is the slowest of the two. It has the advantage, however, of requiring less preprocessing, namely just the creation of the XML input file. During the development of ExcelComp, this XML file was produced by running an Excel macro [8]. It is also easier in this mode for the

client programmer to provide the application user the flexibility to apply ExcelComp to a different spreadsheet by simply changing a filename reference in ExcelComp's constructor. The comprehensive update of all dependent cell values upon the change of a constant cell in this mode allows the ExcelComp user to highlight the newly computed dependents. Such an application was developed by the author, where changed cells are shown in a JTable [22] with changed values highlighted in red to allow the user to gain insight into the impact of the change of a cell value.

The compiled mode is *much* faster than the interpreted mode, and should be preferred in cases where a client demands exceptionally high execution time performance, e.g., providing a real-time screen update that depends on the recomputations. The preprocessing needed for compiled mode includes the generation of Java source code that implements the formulas of the spreadsheet. This source code generation was automated by using a Java class that uses classes produced by parser generators [1, 14] to implement a custom parser for a subset of the Excel formula language. In general, Java classes that represent each cell formula in the spreadsheet must be created, and then be referenced by the classpath option for the Java virtual machine (JVM) [18].

## 4    Computation Model

To provide a lingua franca for the discussion to follow, we need to identify parts of a spreadsheet that are useful to us. While the intent is to have a model that is generic, platform-, and vendor-independent, the use of Excel as a reference implementation for ExcelComp influenced the latter's design. The language of this paper will be similarly influenced, however it is germane to any spreadsheet that adheres to the computation model described here. While a queue-based computation model has been successfully developed [16], we will find it advantageous to develop a model using graph theory. In particular, proof of correctness of the algorithms can benefit from such a treatment.

There are many ways to present data using a spreadsheet. For example, two principal classes of representation provided by Excel are the *workbook*, and the *chart*. We consider only the tabular computation environment found in a workbook. The term *spreadsheet* will thus be used as a synonym for workbook.

A **spreadsheet** is a finite set of cells arranged as a matrix. A **cell** is a set that contains three elements of interest:

1. a value,

2. a formula, and

3. a cell reference.

A cell's **value** is the result of the computation specified by its formula. In general, this value may be a real number, a string, or some other data type. To simplify this model, we will assume that these values are real. A cell's **formula** is an expression that defines a cell's value as a function, $f$, of a subset of the spreadsheet's cell values. Let $\mathcal{C}$ denote the set of cell values for some spreadsheet. More formally then, we have

$$f : \mathcal{C}^n \to \mathcal{C}, \tag{1}$$

where $\mathcal{C}^n$ is the $n$-fold Cartesian product of $\mathcal{C}$ for some positive integer $n$. For this model, we define $\mathcal{C} = \mathbb{R}$. In general, a formula expresses a composite of functions in the form (1). A formula that is not composite has constant values for its arguments. Such a formula is termed a **constant formula**.

A **cell reference** is an ordered pair that specifies a cell uniquely within the spreadsheet. The Excel "A1 reference style"[5] will be used, where the first element specifies the column, and the second element specifies the row. For example, B3 designates the cell at the intersection of the second column and the third row. We use Xi to denote a variable whose value is a cell reference. In the context of a formula, a cell reference is mapped to its corresponding cell value according to (1). We thus see that, in general, a cell reference refers to a composite function that is defined by the formula for that cell. For example, if cell M1 depends on N1 and P2, and N1 depends on Q3, and P2 depends on R1, then the value of M1 expressed as a composite function is M1(N1(Q3),P2(R1)). For this expression to be fully resolved, the formulas for both Q3 and R1 must be constant formulas.

To successfully develop the subject algorithms, the scope of the set of spreadsheets to be considered must be defined. This will be done by identifying properties that serve as axioms for the spreadsheets of interest. Spreadsheets that satisfy the stated properties are termed **admissible**.

There is a cohesive relationship between a cell's formula and its value, as described in the following property.

**Property 1.** *A cell's value is completely determined recursively by its formula.*

To be clear, Property 1 states that a cell's value depends only on its formula, but that formula in general depends on other cell values that are, in turn, dependent only on their formulas. This recursion ends when a cell with a constant formula is reached, thus resolving all of the recursive cell value references.

The formulas of a spreadsheet define a potentially involved relation among its cells. Given cell Xi, its formula may be a constant formula, or a non-constant formula that defines Xi's value as a function of other cell values in the spreadsheet. In the latter case, we say that Xi is **dependent** on the cells referenced in its formula. That is, the *value* of Xi depends on the *values* of the cells referenced in its formula. The term "value" will often be omitted when the context of "dependent" is clear. We also refer to each cell referenced in Xi's formula as a **child** of Xi. Similarly, Xi is a **parent** of its children. A parent and any parent of a parent is termed an **ancestor**. A child and any child of a child is termed a **descendant**.

The relation among the dependent cells in a spreadsheet may be represented as a weakly connected directed graph, $G(V, E)$, where $V$ is the set of vertices, and $E$ is the set of edges. Figure 1 illustrates this **spreadsheet dependency graph** with an example that will be used throughout this paper.
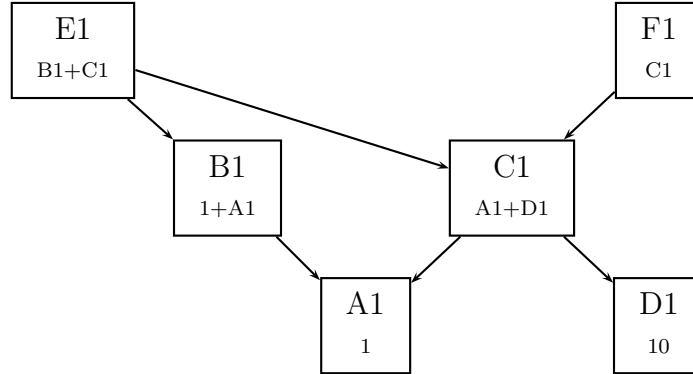


Figure 1: Example of a Spreadsheet Dependency Graph

Each vertex of the graph is a cell represented as a box. The cell reference is given at the top of each cell box, and its corresponding formula is

given in smaller type at the bottom of the box. It is understood that the value of a cell is assigned the value computed by its formula. The cell values are omitted from the cell boxes in Figure 1 for brevity. Note that C1 has two parents: E1 and F1. This lack of a unique parent in general for each cell precludes regarding this structure as a rooted tree. Although the more general *graph* is the appropriate data structure for representing cell dependencies in a spreadsheet, we will find that rooted trees will also be useful in the algorithms to be described.

In describing Figure 1, some of the basics of graph theory, using [6] as a guide, will be described as needed.

In Figure 1, the set of vertices $V$ are the cells, and the set of directed edges $E$ is defined according to the parent/child relationships. Each edge is directed from a parent to a child. The set $E$ is a subset of the ordered pairs of $V$. Let a sequence of vertices be ordered such that $v_{i-1}$ is a parent of $v_i$, and let $v_{i-1}v_i$ denote the edge directed from $v_{i-1}$ to $v_i$. A sequence of edges and vertices that has the form $\{v_0v_1, v_1v_2, \ldots, v_{n-1}v_n\}$ for distinct $v_i$ is defined as a **directed path** linking $v_0$ and $v_n$. A directed path is a **directed cycle** if it consists of 2 or more vertices, and $v_n = v_0$.

We are now led to an important stipulation concerning spreadsheets.

**Property 2.** *An admissible spreadsheet contains no directed cycles.*

The Excel term for directed cycle is **circular reference**. Property 2 thus states that circular references are prohibited.

A graph from the subset of graphs just described is termed a **directed acyclic graph** or **dag** [3, section B.4].

Readers familiar with the GNU Make tool [28] will recognize this dependent cell recomputation problem as being analogous to the problem solved by Make: automatic determination of the pieces of code that require recompilation, and issuing the appropriate commands to bring the program up to date. Make uses a dependency graph model. See [26] for details and illustrations of Make's dependency graphs, including a description of pitfalls concerning the proper use of Make to ensure correct dependency graph construction in large projects.

# 5 Interpreted mode

This section describes those algorithms that are implemented in the interpreted mode of ExcelComp. The integrity of the spreadsheet is preserved in this mode by recomputing all dependent cells of a cell whose constant formula (value) has changed. This behavior ensures that, upon the commitment of a new value to a cell, the entire spreadsheet will be updated to reflect the change. This matches the default behavior of Excel, where it is termed *automatic calculation.* Pseudocode is given in this section to highlight the salient features of the ExcelComp interpreted mode algorithms; the actual code differs in some of the implementation details.

## 5.1 Dependency Set Generation

Suppose that we are examining a spreadsheet for the first time, and have no a priori knowledge of its contents. Say we want to modify cell A1. By this, we mean that A1 has a constant formula that is to be changed to another constant formula. The more general act of modifying or adding a non-constant formula will not be discussed here; it is assumed that non-constant formulas remain fixed throughout our analysis.

Consider the impact that this change has on the cells that are dependent on A1 in Figure 1. First, this change in the formula causes a recomputation of A1's value. This change will, in general, affect the values of all cells whose formulas reference A1. These cells, B1 and C1, are **directly dependent** on A1, and will need to be recomputed as a result. In general, the values of these direct dependents will change as a result of recomputation. These direct dependents must then be considered in the same light as A1; that is, we need to find and recompute the direct dependents of the direct dependents of A1. These cells are E1 and F1. From the point of view of A1, these latter cells are **indirect dependents** of A1.

The algorithm for discovering the set of dependent cells of a given cell is thus recursive. Let $d$ be a set-valued function $d : 2^{\mathcal{C}} \to 2^{\mathcal{C}}$ that computes the set of direct dependents of a subset of cells from $\mathcal{C}$. (Here, $2^{\mathcal{C}}$ denotes the set of all subsets of $\mathcal{C}$, also known as the *power set.*) The procedure just described can now be expressed as

$$A_{i+1} = d(A_i), \qquad A_i \in 2^{\mathcal{C}}, \qquad A_0 = \{\text{A1}\}. \tag{2}$$

$A_0$ is set to {A1} in (2) to reflect Figure 1, but in general it will be assigned to the set containing the cells that were changed.

Dependency Set Generation may be recognized as an implementation of the *breadth-first search* (BFS) algorithm for graphs described in [3]. The "frontier between discovered and undiscovered vertices" described in [3] applied here divides two generations of dependencies, i.e., child/parent, parent/grandparent, etc. Also, note that we begin with a child, and then discover ancestors, in reverse to the naming convention used in [3].

For recurrence relation (2) to be practical, we must be assured that it terminates. Indeed, an essential property of an algorithm is its finiteness; according to [15], "An algorithm must always terminate after a finite number of steps." This assurance is given now as a theorem.

**Theorem 1.** *Recurrence relation* (2) *terminates.*

*Proof.* Because the indices of $A_i$ in the recursion are strictly increasing, it is sufficient to show that $\exists i_{max} \ni i \leq i_{max}$.

Assume that the spreadsheet under consideration has $N$ cells. Let $\|A_i\|$ denote the number of parents of $A_{i-1}$. Because of Property 2, the number of candidate parents for $A_0$ is $N - 1$, since a cell cannot be a parent of itself (thereby creating a circular reference). Similarly, the number of available parents for $A_1$ is at most $N - 2$, since both the children and grandchildren of $A_2$ must be excluded to avoid circular references. In general then,

$$\|A_i\| \leq N - i,$$

and in particular, $\|A_N\| = 0$. At this point, no parents are available to continue further. We have thus shown that $i \leq N$; that is, $i_{max} = N$. □

The final product of Dependency Set Generation is formed by taking the union of the sets of dependent cells found in (2). Assuming that the final index computed in (2) is $n$, and letting $\mathcal{D}$ be the set of dependent cells, we have

$$\mathcal{D} = \bigcup_{i=1}^{n} A_i, \qquad A_i \in 2^{\mathcal{C}} \tag{3}$$

```
Input: Set of cells for which dependent cells will be found
Output: Set of cells that are dependent on the input
        set of cells
DepSetGen(depSet,m)
{
  initial_size = depSet.size();
  for (k=m through initial_size-1;k++) {
    for (j=0 through SPRSHEET_SIZE-1;j++) {
    // Find all direct dependents of depSet[k].
      if (sprsheet[j].formula contains depSet[k].ref) {
        depSet.add(sprsheet[j].ref);
      }
    }
    DepSetGen(depSet,initial_size);
  }
  if (m == 0) {
    depSet.delete(depSet[0]);
  }
}
```

Figure 2: Dependency Set Generation Algorithm

### 5.1.1 Example

The Dependency Set Generation algorithm is codified in Figure 2. Let us apply this algorithm to finding all dependents of A1 in Figure 1.

Assume the number of cells in the spreadsheet, SPRSHEET_SIZE, is 6. The array sprsheet holds all the cells in the spreadsheet. Each element of sprsheet has the fields ref and formula for cell reference and formula, respectively. The array depSet will be built up to contain the cell references of all of the dependents of its initial value.

We make the initial call to DepSetGen with depSet initialized to contain A1, and the depSet element marker m set to 0. This marker's value is the index of the cell in array depSet whose dependents are sought. Variable initial_size is set to the number of elements in depSet. Since depSet contains only A1, initial_size = 1. Loop counter k ranges from 0 through 0. The inner loop checks to see whether any spreadsheet cell formula contains

A1. If it does, the cell reference is added to `depSet`. At the time where the inner loop is finished, both B1 and C1 are appended to `depSet`. At the bottom of the outer loop, it is time to make the first recursive call to `DepSetGen`.

The actual parameters passed to `DepSetGen` are the newly updated 3-element `depSet`, and the initial size of `depSet` before the loops, 1. Now entering the first recursive call of `DepSetGen`, `m` is 1, `initial_size` is 3, `depSet` consists of A1, B1, and C1. Loop variable `k` ranges from 1 through 2. The first time through the inner loop finds all dependents of B1, and adds the one dependent found, E1, to `depSet`. When `k` is 2, the inner loop finds all dependents of C1, and adds the one dependent found, F1, to `depSet`.

Upon the next recursive call, the marker is set to the next unexamined element of `depSet`, E1 at index 3. No dependents are found. Similarly for F1 at index 4. Finally the outer loop is skipped, and control is eventually returned to the original call of `DepSetGen`, where `m` is 0. Lastly, the `if` statement is executed, and the initial element A1 is removed from `depSet`, since A1 is not dependent on itself. It is assumed `depSet` contains just one cell during the initial call to `DepSetGen`. Though not shown in Figure 2, an additional step of removing duplicate cell references from $\mathcal{D}$ is required to ensure that all of its elements are unique.

To conclude this section, it will be shown that the Dependency Set Generation algorithm just described indeed finds all dependents of a given cell.

**Theorem 2.** *The Dependency Set Generation algorithm identifies all dependents of its input set of cells.*

*Proof.* The proof is by contradiction. Assume we have a spreadsheet with cells $C_i$, $i = 0, 1, 2, \ldots, N-1$. Suppose $\exists C_k \in \mathcal{C}$ that is dependent on $C_0$, but was not identified by the Dependency Set Generation algorithm. Then by Property 1, this dependence of $C_k$ on $C_0$ must be due only to $C_k$'s recursive formula. There must then be a directed path in $C_0$'s dependency graph from $C_k$ to $C_0$. Since $C_k$ was not identified, there is at least one cell $C_j$ in this path that was not found during the recursion. But this contradicts the step in the algorithm that says to find *all* direct dependents of $C_{j-1}$. $\square$

## 5.2   Recomputation of Dependency Set Members

Once the dependency set $\mathcal{D}$ has been generated, the process of recomputing these cells can begin.

To evaluate a cell, designated the **original cell**, each argument in its formula, the **original formula**, must be evaluated. This evaluation is in general a recursive procedure. By drawing a directed edge from the original cell to each cell referenced in the formula (one edge per cell in the formula), we get a rooted tree that is rooted at the original cell. By applying this algorithm recursively on each cell in the formula, we get several paths, each of which ends at a leaf having a constant-formula cell. The resulting rooted **call tree** is just a subset of the spreadsheet dependency graph. We may then start at the leaves of the tree to construct a string of formulas of the cells in a given path in the direction back toward the root. Each string is a self-contained sequence of formulas that allows the original cell to be evaluated.

Lastly, once all the arguments in the original formula have been evaluated, the original cell can be evaluated. Those arguments in any formula that are not members of $\mathcal{D}$ need not be recomputed; their current values can be used instead.

The foregoing procedure is an implementation of the *depth-first search* (DFS) algorithm described in [3]. In this case, the use of the terms "predecessor" and "descendant" in [3] is consistent with our usage. However, we do not use "timestamping."

We can see a lot in common here with the Dependency Set Generation algorithm. Once again, we see a recursive procedure being described, although it is not as easily expressible in one line as in (2). Instead, we will codify the algorithm in the pseudocode given in Figure 3.

### 5.2.1   Example (continued)

Continuing the example begun in the Dependency Set Generation section, consider again the dependency graph in Figure 1. Suppose that E1 is to be evaluated. The well-known *left-to-right, post-order tree traversal* algorithm will be used to specify the order of evaluation of E1's descendants. During the first call to `EvalCell`, the recursion depth variable `depth` is initialized to 0. In addition to the `ref` and `formula` fields, assume that a `cell` object also contains a field `nchildren` that gives the number of children in its `formula` field. The object `cell` also contains a `child` array, each of whose elements is a `cell` representing each child in its formula. The elements of `child` are stored in order of occurrence in its `formula`, element 0 being the leftmost child, and element `nchildren-1` being the rightmost.

```
Input: Original Cell
Output: Original Cell with newly computed value
depth = 0;  // Initial value (global)
EvalCell(cell)
{
  for (i=0 through cell.nchildren-1;i++) {
     depth++;
     EvalCell(cell.child[i]);
  }
  parser_str.append(cell.ref + '=' + cell.formula + ';');
  if (depth == 0) {
    cell.value = parse(parser_str);
  }
  depth--;
  return cell;
}
```

Figure 3: Dependency Set Evaluation Algorithm

We thus begin by calling `EvalCell` with `cell.ref` set to E1. The recursive evaluation of E1's call tree begins with the leftmost cell reference in E1's formula, B1. B1 $\in \mathcal{D}$, and therefore must be recomputed. We then begin with the leftmost element of its formula, and find that it is the constant 1. This is a constant, and thus requires no further evaluation; we move to the next element, A1. A1 is the changed cell, and its value is known, thus no further analysis is needed.

We have now reached the end of B1's formula, allowing us to compute its value. We thus go back up to E1 to process the next argument in its formula, C1. C1 $\in \mathcal{D}$, and therefore must be recomputed. The leftmost child of C1's formula is A1, and has already been evaluated. C1's next child, D1 $\notin \mathcal{D}$, and thus does not have to be recomputed. Its current value of 10 is used.

We have now recursively evaluated all of the children of E1's formula, and completed the building of `parser_str`. This string may then be passed to a parser for evaluation of E1. ExcelComp uses an LALR parser developed by the author using the tools JLex [1] and CUP [14]. LALR stands for LookAhead Left-to-right identifying the Rightmost production, and is described in [12]. For this example, the string is:

13

```
A1=2;B1=1+A1;A1=2;D1=10;C1=A1+D1;E1=B1+C1;.
```

Here, "=" stands for assignment, and ";" delimits each assignment. It is assumed that the parser stores values via the assignment statements, and that these values may be retrieved at points later in the parse string. The history of the construction of the parser string is summarized in Table 1.

| depth | cell.ref | i | parser_str |
|---|---|---|---|
| 0 | E1 | 0 | |
| 1 | B1 | 0 | |
| 2 | A1 | 0 | A1=2; |
| 1 | B1 | 0 | A1=2;B1=1+A1; |
| 0 | E1 | 1 | A1=2;B1=1+A1; |
| 1 | C1 | 0 | A1=2;B1=1+A1; |
| 2 | A1 | 0 | A1=2;B1=1+A1;A1=2; |
| 1 | C1 | 1 | A1=2;B1=1+A1;A1=2; |
| 2 | D1 | 0 | A1=2;B1=1+A1;A1=2;D1=10; |
| 1 | C1 | 1 | A1=2;B1=1+A1;A1=2;D1=10;C1=A1+D1; |
| 0 | E1 | 1 | A1=2;B1=1+A1;A1=2;D1=10;C1=A1+D1; E1=B1+C1; |

Table 1: History of Parser String Construction for E1

Note that there is a redundant assignment "A1=2" in `parser_str`. This is an example suggesting efficiency enhancements that can be seen to improve the performance of these algorithms. Improvements include, but are not necessarily limited to:

1. Recompute a cell value only once.

2. Do not recursively evaluate cells that are not in $\mathcal{D}$.

Although these improvements surely are desirable for minimizing the number of algorithm steps, experience with their use in ExcelComp revealed that the time to run the additional code required to implement these improvements largely cancels out the benefits of fewer cell evaluations.

More ideas concerning the speed-up of interpreted mode are given in the Algorithm Complexity section.

# 6 Compiled mode

While the compiled mode agrees with interpreted mode with respect to the preservation of spreadsheet integrity, it uses a clever postponement of computation technique to update a dependent cell's value just prior to the reading of its value via its accessor method. Using this deferred recomputation strategy, the execution time associated with recomputing those dependent cells whose values are never accessed is eliminated. This deferred recomputation is similar to Excel's *manual calculation* mode, where the user specifies when recomputation is to occur thus deferring immediate recomputation. It is also similar to the *mark-sweep* garbage collection algorithm [29].

Compiled mode is implemented by ExcelComp's use of the `Cell` API [27]. The highlight of this mode is preserving spreadsheet integrity while improving the running time performance of ExcelComp. Several techniques are used to meet this goal, including the use of:

1. the Java Reflection API

2. Hash containers, and

3. Deferred recomputation of dependent cells.

## 6.1 The Cell Class

`Cell` is an abstract Java base class that provides a framework for modeling the cells of a spreadsheet, each of which is represented by a class derived from `Cell` named `CellXi`, where `Xi` denotes the A1-style reference to its corresponding cell.

On its initial invocation, `Cell`'s accessor class method `getCell` instantiates a `CellXi` object via the Java Reflection API [25]. This technique allows a Java class to instantiate another class whose name is created at run time by the calling method. To improve the efficiency of subsequent accesses, `Cell` has a class variable `workbook` to reference a `HashMap` of references to previously instantiated `CellXi` objects. Similarly, each `CellXi` object has a `HashSet` named `dependencies` that contains references to `CellXi` objects that correspond to cells that are direct dependents (parents) of the `CellXi` object that owns `dependencies`.

In a `CellXi`'s constructor, each child's instance method `addDependency` is called to add Xi to that child's `dependencies` set. The use of `getCell` during
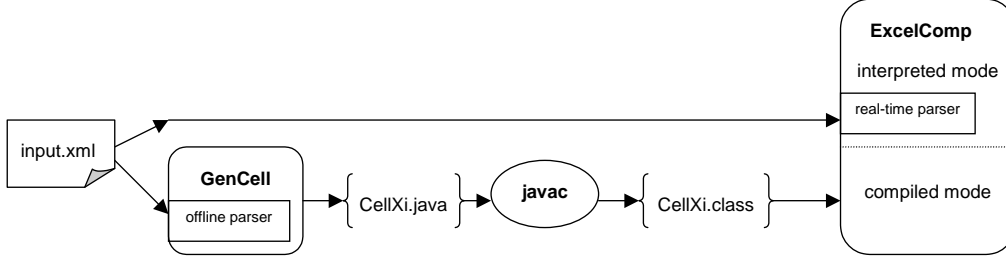
Figure 4: Preprocessing requirements for ExcelComp modes

this process causes each child's object, on its first access, to be initialized with its constructor. A DFS traversal of `CellXi`'s call tree thus occurs so that each traversed parent appears in each of its children's `dependencies` set. The set of all such parents is the set of **discovered ancestors**. Both `workbook` and `dependencies` contain only minimal subsets of the full set of their respective data that describes the entire spreadsheet as determined by the history of `Cell` method calls. These subsets are updated as necessary, and suffice for computing correct results when recomputation of cell values is necessary.

## 6.2 Preprocessing

Figure 4 details the differences in the preprocessing requirements for the two modes of ExcelComp.

The preprocessing necessary for compiled mode begins with the same XML input file used in interpreted mode. A Java API named `GenCell` [10] is run on this input file to produce a set of Java source files. Each source file defines a `CellXi` class that corresponds to a spreadsheet cell. `GenCell` uses a JLex/CUP-based parser similar to that used in interpreted mode to allow the translation of a supported subset of the Excel formula language into the appropriate Java statements. The important difference between the two parsers is in their output. In interpreted mode, the ExcelComp parser returns a string at run time that represents a newly-computed value. For ExcelComp's compiled mode, the `GenCell` parser returns Java source code that is written to a set of source files as an offline preprocessing task. Once `GenCell` has completed generating all of the source files, the Java compiler is run to compile these files into class files of executable bytecode. The Excel-Comp user must then ensure that the Java classpath contains the appropriate

16

references to allow the client application to find the classes at run time.

## 6.3   Dependency Set Generation

The nature of the preprocessing performed in compiled mode allows Excel-Comp to handle Dependency Set Generation as a distributed, on-demand task rather than as an explicit set of steps conducted immediately after a cell value is changed as in interpreted mode.

## 6.4   Deferred Recomputation of Dependent Cells

Rather than requiring that the immediate recomputation of all members of $\mathcal{D}$ occur after a cell's value has been changed, compiled mode defers the recomputation of any member of $\mathcal{D}$ until the client programmer requests its value via an ExcelComp accessor method. This algorithm saves considerable time when compared to its interpreted mode counterpart, seeing that the recomputation of many dependencies whose value is never sought is avoided.

## 6.5   Getting a Value

Each `CellXi` object's `getValue` method contains the Java encoding of the formula for cell Xi. The value of the cell is recomputed and stored as the instance variable `val` only when that object's boolean `dirty` flag is true; this flag thus allows this method to avoid unnecessary recomputation. When recomputation is unnecessary, `getValue` just returns the value of `val`.

## 6.6   Setting a Value

When a `CellXi` object's `setValue` method is called, its instance variable `val` is set to the desired value, and a DFS traversal of all of Xi's discovered ancestors is performed to ensure that each such ancestor's `dirty` flag is set to `true`. This ensures that all ancestors' values are recomputed on a subsequent call to an ancestor's `getValue` method. Note that recomputations are only done if an ancestor's `getValue` method is called, thus saving the time of recomputing ancestor values that may never be accessed.

## 6.7 Example

The following example illustrates the workings of compiled mode using Figure 1. Given the spreadsheet represented by this graph, we will use compiled mode to set A1's value to 2, and then get the value of E1. Observe that E1's value with A1 set to 1 is 13. By changing A1's value to 2, we expect the new value of E1 to be 15.

The first statement given by the client program is:

```
Cell.getCell("A1").setValue(2);
```

The initial part of this statement, `Cell.getCell("A1")`, creates a new instance of `CellA1`, since one does not yet exist. This new instance is created using the `forName` class method in the package `java.lang.Class` [19]. `CellA1`'s base class constructor `Cell()` is first called to set `CellA1`'s `dirty` flag to `true`. `CellA1`'s `val` variable is set to 1 by its constructor. The last part of this statement calls `Cell`'s `setValue` method to set `val` to 2. In general, `setValue` recursively marks all of A1's *discovered* ancestors as dirty. However, since no ancestors have yet been discovered, and thus no corresponding `CellXi` objects have yet been instantiated, no such marking occurs here. We are thus left with one instance of `CellA1` that is marked as dirty and has a value of 2.

We now get the value of E1. The appropriate statement is:

```
Cell.getCell("E1").getValue();
```

The first part of the statement behaves in the same way as for `A1` described above, only now the newly created object is an instance of `CellE1`. In addition, `CellE1`'s constructor initiates a DFS traversal of all of E1's descendants to update each descendant's `HashSet dependencies`. For this example, both `CellB1`'s and `CellC1`'s `dependencies` sets are updated by having a reference to `CellE1` added. Note that `CellC1`'s `dependencies` set does not refer to `CellF1` since, although F1 is dependent on C1, it is not a descendant of E1. Such a reference to `CellF1` need only be added if F1 is the subject of future method calls. The last part of this statement checks to see whether the instance of `CellE1` has been marked as dirty. Since `CellE1` was just constructed anew, it is marked as dirty and thus its value must be recomputed (in this case, computed for the first time). The statement in `CellE1`'s `getValue` method that accomplishes this is:

```
this.val = Cell.getCell("B1").getValue() +
           Cell.getCell("C1").getValue();
```

Since the objects for B1, C1, and D1 were all marked as dirty during the DFS traversal in `CellE1`'s constructor, each object's `getValue` method will recompute the value for that cell. Each object's `getValue` method resets that object's `dirty` flag to `false` after the recomputation. Subsequent accesses to `CellXi` values that have not been affected by a change to a descendant's value simply return the value stored in `val` with no recomputation necessary.

# 7  Performance

## 7.1  Tests

ExcelComp was tested for its running time performance in the SATCOM Availability Analyst (SA2) Java application [7]. The addition of ExcelComp API calls to SA2's map display function was chosen for test due to its demanding requirement that 8,518 data points be updated for an on-screen Mercator map in such a way that the user is not burdened by long wait times for a complete update of the map. Processing each of the data points required 3 calls to ExcelComp methods; 2 of these calls each changed a cell value from the input spreadsheet, and the last call read back a cell value of interest from the newly updated spreadsheet. The map display function was selected and run 10+ times in each mode to characterize ExcelComp's performance. Running times associated with the first invocation of the map function were greater than subsequent trials, and thus were considered outliers and removed from the representative data. These larger values probably reflect JVM-related setup steps that are not required on subsequent trials.

The tests were conducted on a Hewlett-Packard HP OmniBook 4150 B running under Microsoft Windows 98 on a Pentium III 650 MHz processor. SA2 was run using the Sun Microsystems JVM version 1.3. Running times were computed as the difference in the start and end times returned by the Java method `System.currentTimeMillis()` [23].

The results of the performance tests are summarized in Table 2. The sample standard deviation is computed as the positive square root of the *unbiased* sample variance. The large difference in performance between the modes highlights how compiled mode can provide a very acceptable performance level in a case where interpreted mode, requiring over 5 minutes to

|                           | Compiled Mode | Interpreted Mode |
|---------------------------|---------------|------------------|
| Average                   | 677           | 347057           |
| Sample Standard Deviation | 116           | 254              |

Table 2: ExcelComp Running Time (milliseconds) Performance over 10 Samples

complete, would be unacceptably slow. In this particular case, it is essential that compiled mode be chosen to make the use of ExcelComp feasible.

## 7.2 Algorithm Complexity

The graph traversal algorithms that underlie ExcelComp are well known to be efficient. Both DFS and BFS have running times that are linear in the size of the graph's adjacency list. Specifically, BFS is $O(V + E)$, and DFS is $\Theta(V + E)$ [3, sect. 22.2, 22.3].

Interpreted mode does not construct an adjacency list, and could very well benefit from a redesign to create this list upon the loading of the spreadsheet in the ExcelComp constructor. Because this construction must take place at run time, the user would incur a one-time performance penalty for this initialization step. The absence of an adjacency list suggests that interpreted mode's running time is probably greater than the linear time cited above.

Compiled mode, on the other hand, does use a variation of adjacency lists in `CellXi`'s `dependencies` set. However, while an adjacency list stores references to children, `dependencies` stores references to parents of `CellXi`. Compiled mode incurs a setup penalty during the discovery of cells, but subsequent accesses to `CellXi` objects are more efficient through the use of `dependencies` and `workbook`. Its use of adjacency lists, DFS traversals, and the efficient Java collections framework suggests that compiled mode has a running time that is close to $\Theta(V + E)$.

## 8 Conclusion

We have seen that a graph representation of spreadsheet cell dependencies provides insight into the requirements of the algorithms used for the automatic recomputation of dependent cells. Straightforward implementation

of well-known graph traversal algorithms suffices for correct recomputation, however the adaptation of a well-studied garbage collection algorithm along with facilities made available in the Java language enable client programs to run much faster, given some additional preprocessing.

The client programmer should choose the mode of ExcelComp according to an analysis of the application's run time requirements and the tradeoffs between the modes as described in this paper.

# 9    Acknowledgements

I would like to thank Deborah Schuh and Dr. Joseph Rushanan for their reviews and helpful comments on the contents of this paper.

# References

[1] Elliot Berk. JLex: A Lexical Analyzer Generator for Java™. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

[2] Tim Bray et al. Extensible Markup Language (XML) 1.0 (Second Edition). `http://www.w3.org/TR/REC-xml`.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 2d. edition, 2001.

[4] Microsoft® Corporation. Microsoft® Office - Excel Home Page. `http://www.microsoft.com/office/excel/default.asp`.

[5] Microsoft® Corporation. *Microsoft® Excel 97 Help*, 1997.

[6] R. Diestel. *Graph Theory*. Springer-Verlag, 2000.

[7] Gerry Fitzgerald. SATCOM Availability Analyst (SA2). The MITRE Corporation.

[8] Joe Francoeur. Excel-to-XML macro. The MITRE Corporation.

[9] Joe Francoeur. ExcelComp - a Java™ class for automated recomputation of spreadsheets. The MITRE Corporation.

[10] Joe Francoeur. GenCell - a Java™ class for generating Cell-related classes. The MITRE Corporation.

[11] Joe Francoeur. TestExcelComp - a Java™ class for testing ExcelComp. The MITRE Corporation.

[12] D. Grune and C. J. H. Jacobs. *Parsing Techniques - A Practical Guide.* World Wide Web, `http://www.cs.vu.nl/~dick/PTAPG.html`, 1998.

[13] J. Hopcroft, J. Ullman, and R. Motwani. *Introduction to Automata Theory, Languages and Computability.* Addison Wesley Longman, Inc., 2001.

[14] Scott E. Hudson. CUP Parser Generator for Java™. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[15] D. E. Knuth. *The Art of Computer Programming.* Addison Wesley Publishing Co., Inc, 2d. edition, 1973.

[16] Loreen La Penna. Recalculation in Microsoft® Excel 2002. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexcl%2k2/html/odc_xlrecalc.asp`, 2001.

[17] Michael Meeks. A discussion of the new dependency code, version 0.2. Available in dependencies.txt in gnumeric-1.0.9.tar.gz at `http://www.gnome.org/projects/gnumeric/index.shtml`.

[18] Sun Microsystems, Inc. `java` - the Java application launcher. `http://java.sun.com/j2se/1.3/docs/tooldocs/PLATFORM/java.html`, where PLATFORM = solaris, linux, or win32.

[19] Sun Microsystems, Inc. Java™ 2 Platform SE v1.3.1: Class Class. `http://java.sun.com/j2se/1.3/docs/api/java/lang/Class.html`.

[20] Sun Microsystems, Inc. Java™ 2 Platform SE v1.3.1: Class HashMap. `http://java.sun.com/j2se/1.3/docs/api/java/util/HashMap.html`.

[21] Sun Microsystems, Inc. Java™ 2 Platform SE v1.3.1: Class HashSet. `http://java.sun.com/j2se/1.3/docs/api/java/util/HashSet.html`.

[22] Sun Microsystems, Inc. Java™ 2 Platform SE v1.3.1: Class JTable. `http://java.sun.com/j2se/1.3/docs/api/javax/swing/JTable.html`.

[23] Sun Microsystems, Inc. Java™ 2 Platform SE v1.3.1: Class System. `http://java.sun.com/j2se/1.3/docs/api/java/lang/System.html`.

[24] Sun Microsystems, Inc. Java™ 2 Platform, Standard Edition (J2SE™). `http://java.sun.com/j2se/1.3/`.

[25] Sun Microsystems, Inc. The Reflection API. `http://java.sun.com/docs/books/tutorial/reflect/index.html`.

[26] Peter Miller. Recursive Make Considered Harmful. Available at `http://www.canb.auug.org.au/~millerp/`.

[27] Paul Silvey. Class Cell. The MITRE Corporation.

[28] Richard M. Stallman and Roland McGrath. GNU Make - A Program for Directing Recompilation. `http://www.gnu.org/manual/make-3.79.1/html_chapter/make_toc.html`.

[29] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. `ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`.